VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

# Absolvování individuální odborné praxe

Individual Professional Practice in the Company

Jakub Čačka

Bakalářská práce

Vedoucí práce: Ing. Jan Kožusznik, Ph.D.

Ostrava, 2021

## Abstrakt

Tato bakalářská práce pojednává o průběhu absolvování individuální odborné praxe ve spolčenosti Moravio s. r. o. První část se zabývá firmou jako takovou, jejím technickým zaměřením a projektem, ke kterému jsem byl přiřazen. Druhá část popisuje úkoly, které jsem na projektu zpracovával, jejich zadání, řešení a výsledky. V poslední kapitole zhodnotím využité dovednosti nabyté při studiu a praxi.

## Klíčová slova

odborná praxe, Moravio s. r. o., PHP, Laravel, JavaScript, MySQL, HTML, Docker, Google Cloud

## Abstract

This thesis describes the process of completing individual professional practice in the company Moravio s. r. o. The first part of this document describes the company, it's environment, technical classification and the project, to which I was assigned. The second part is about some of my tasks on that project, along with chosen solution, procedure and results. The last part of this thesis is comparing experience and skills acquired in my studies and during the practice.

## Keywords

Professional Practice, Moravio s. r. o., PHP, Laravel, JavaScript, MySQL, HTML, Docker, Google Cloud

## Poděkování

Rád bych zde poděkoval své rodině za podporu při studiu, kolegům za příjemně strávený čas ve firmě a ochotu, a také vedoucímu mé práce za pomoc při její tvorbě.

# Obsah

# Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| AJAX | – | Asynchronous JavaScript And XML |
| API | – | Application Programming Interface |
| CI/CD | – | Continuous Integration/Continuous Deployment |
| CRC | – | Cyclic Redundancy Check |
| CRUD | – | Create, Read, Update, Delete |
| CSS | – | Cascading Style Sheets |
| DB | – | Database |
| DevOps | – | Development and Operations |
| GCP | – | Google Cloud Platform |
| HTML | – | Hypertext Markup Language |
| IT | – | Information Technology |
| JS | – | JavaScript |
| JSON | – | JavaScript Object Notation |
| MVC | – | Model-View-Controller |
| NPM | – | Node.js package manager |
| SQL | – | Structured Query Language |
| SSH | – | Secure Shell |
| YAML | – | YAML Ain't Markup Language |

# Seznam obrázků

# Seznam tabulek

# Kapitola 1

# Úvod

Jako téma své bakalářské práce jsem si vybral absolvování individuální odborné praxe. Hlavním důvodem této volby byla potřeba získání zkušeností nejen v okruhu programování, ale také v oblasti chodu firmy a trhu práce. Se společností Moravio s. r. o. jsem se díky studijního plánu univerzity seznámil již ve čtvrtém semestru v předmětu Praxe. Už v té době jsem nahlédl pod pokličku procesů firmy a jejích projektů, vyzkoušel si práci v týmu, měl možnost být v kolektivu zkušenějších programátorů a zjistil, jak moc přínosná může být jejich přítomnost. Obsah této práce je rozdělen do tří částí. První část se zabývá firmou jako takovou, jejím technickým zaměřením a projektem, ke kterému jsem byl přiřazen. Druhá část popisuje úkoly, které jsem na projektu zpracovával, jejich zadání, řešení a výsledky. V poslední kapitole zhodnotím využité dovednosti nabyté při studiu a praxi.

# Kapitola 2

# Popis firmy Moravio s. r. o.

Společnost založil Lukáš Greň v únoru 2011. Zpočátku se jednalo pouze o firmu obstarávající vývoj webových prezentací. Později se ale Moravio rozrostlo o další firmu, a to Moravio Online Marketing s. r. o., která se soustředí na zvyšování povědomí o Moraviu jako takovém, své služby ale také nabízí mnohým dalším klientům. Společnost se každým rokem rozrůstá, v současné době zaměstnává téměř 30 lidí.
Moravio je také jedním z hlavních pořadatelů, konferencí ze světa internetu, ostravského Barcampu. Toho se zúčastní nespočet odborníků a začátečníků v oblastech IT, kteří si navzájem vyměňují zkušenosti a znalosti. Poslední ročník se dokonce konal v Aule VŠB-TUO Ostrava.

## 2.1 Technologické zaměření

Firma se ze startu věnovala tvorbě prezentačních webů stavěných na PHP pro společnosti z Ostravy a okolí, z těch zvučnějších například Sareza, Ridera a Veolia. Později začala vytvářet i složitější weby a informační systémy na PHP frameworku Laravelu. Jedním z takových projektů byl informační systém pro správu brigád pro slovenskou společnost Junglee. Mimo to měla ve správě i několik e-shopů, a to například projekty společností Bushman, Dáme Jídlo a Jelínek.

Od roku 2018 se firma rozrostla a začala dělat pro větší české jména, například Mall.cz, ale také i pro společnosti zahraniční, jako třeba nadnárodní JLL. Právě pro JLL pomáhalo vytvářet virtuálního asistenta Jet, který za pomocí hlasové konverzace usnadňuje a vyřeší každodenní úkoly v pracovním prostředí, například efektivním rezervováním kancelářských prostorů. Na tomto projektu Moravio využívá moderní technologie jako Machine Learning, Dialogflow a natural language processing.

V současnosti je firma v procesu transformace využívaných technologií. PHP a frameworky na něm postavené nahrazují JS technologie a jejich frameworky. Seznam technologií, které využívá:

- Node.js

- TypeScript
- React
- Vue.js
- Docker
- Kubernetes
- Google Cloud
- Gitlab CI/CD
- Python
- PHP

Back-end psaný v Node.js a TypeScriptu se stává novým standardem firmy. Front-end technologie vybírá podle potřeby klienta, často však bývá použit React. Stále ale má ve správě weby, které jsou psány v PHP. Také projekt, který je předmětem téhle bakalářské práce, je ještě psán v PHP.

Firma se ale také rozvíjí v DevOps, tedy ve vylepšování procesu vývoje aplikací a jejich nasazování. Dalším standardem se tedy stává využívání kontejnerů Dockeru, nasazování aplikací přes Gitlab CI/CD a jejich provoz na Google Cloud s použitím orchestrace virtuální kontejnerů pomocí nástroje Kubernetes.

## 2.2 Prostředí

Moravio poskytuje práci stále rostoucímu počtu lidí, v současné době jich čítá téměř 30. Jedná se převážně o seniorní programátory, projektové manažery a administrativní pracovníky. Součástí jsou ale také juniorní vývojáři, kteří ještě studují. Díky skvělé interní komunikace firmy jsem měl kdykoliv možnost konzultace se seniorním programátory. Přístupný nám byl i majitel Moravia, který se znalostně mezi seniorní vývojáře rozhodně řadí.

Pro komunikaci firma využívá aplikaci Slack, která umožňuje soukromé psaní a hovory, dále také komunikaci ve vytvořených skupinách. Díky této komunikaci byl člověk opravdu vtažen do dění firmy a stal se tak její součástí. Zároveň díky tomu byl prožitek z praxe velmi dobrý, jelikož se konala v letech 2020 a 2021, kdy byly zavedeny mimořádné opatření a praxi jsem z toho důvodu nemohl z většiny konat prezenčně.

Firma klade důraz na uchovávání interních informací o projektech a jejich vývoji, proto jsme všechny naše práce na projektu zapisovali do dokumentace a jakékoliv informace o obecnějších problémech nebo technologiích jsme sepisovali do firemní dokumentace vedené v Confluence.

V průběhu mé praxe byl v Moraviu zaveden každotýdenní celofiremní status nazvaný LevelUp. Ten měl dva hlavní cíle, zvýšit interní informovanost o stavu a vizi společnosti a sdílení zkušeností mezi kolegy.
V druhém případě na statusu většinou vystoupil jeden z vývojářů, který ostatním popisoval některou ze zajímavých technologií nebo služeb, které v dané chvíli využíval. Mimo to zde také prezentovali zkušenější programátoři, kteří předávali své zkušenosti pro řešení některých problémů.
Sám jsem na LevelUpu jednou vystoupil s prezentací o technologii Sphinx, která v této práci bude ještě popsána.

### 2.2.1 Agilní vývoj

Na projektu praktikujeme agilní vývoj. Ten spočívá v tom, že se na začátku spolupráce s klientem neplánuje vývoj celého projektu a nenaceňuje se celý jeho vývoj. Místo toho se vývoj plánuje po tzv. sprintech, což je stanovený časový úsek, po kterém se budou vyvíjet domluvené funkcionality. My jsme na projektu měli například týdenní sprinty. Klient si neplatí konkrétní úpravu, ale platí si kapacitu vývojáře.

Tento typ vývoje je velice flexibilní. Nedílnou součástí agilního vývoje je úzká komunikace s klientem, které jsem také byl součástí. Dalším prvkem tohoto vývoje jsou pravidelné statusy, v průběhu a také na konci sprintu, kdy se probírá další sprint a dochází zde také k retrospektivě nad provedenými pracemi, aby se předešlo chybovosti v dalších sprintech a lépe se odhadly.

Pro přehlednou správu sprintů a jednotlivých úkolů jsme používali ticketovací systém Jira. Každý úkol jsme si tam rozpracovali do více podúkolů, které se pak snadněji odhadovaly a plánovaly. Reálný čas strávený na úkolech jsme si měřili pomocí aplikace Toggl. Jira zároveň byla napojená na Git projektu a bylo v ní zavedeno spoustu automatizací, které proces vedení sprintů značně ulehčovaly.

## 2.3 Má úloha

V rámci bakalářské praxe jsem se dostal do role PHP developera a spolu s kolegy jsme v agilním vývoji posouvali rozsáhlý projekt Spanamo, který Moravio převzalo v rozpracovaném stavu, k úspěšnému spuštění.

# Kapitola 3

# Použité technologie

V téhle sekci popíši jaké technologie jsem využil v rámci mé praxe.

## 3.1 PHP

PHP je open-source skriptovací programovací jazyk, jehož skripty se spouští na straně serveru (například Nginx, který je popsaný níže). Je určený především pro programování dynamických internetových stránek a webových aplikací.

Je možné jej vložit přímo do HTML kódu pomocí značek `<?php ?>`, mezi které píšeme PHP kód, soubor ale musí mít příponu *.php*. Mezi další důležité vlastnosti jazyka patří, že je dynamicky typovaný, tzn. nemusíme specifikovat datový typ proměnné, ten se specifikuje hodnotou, kterou do ní přiřadíme. Datové typy ale můžeme definovat ve vstupních a výstupních parametrech funkcí. Další specifikací jazyka jsou jeho pole. Pole jsou ve skutečnosti hašovací tabulky, ve kterých jsou uložené páry `klíč -> hodnota`. Klíče mohou být číselné i řetězcové.

## 3.2 MySQL

MySQL je open-source multiplatformní databáze. Jedná se o dialekt SQL jazyka. Chlubí se hlavně snadnou implementovatelností a rychlostí.

## 3.3 HTML

HTML je značkovací jazyk používaný pro tvorbu webových stránek a aplikací, který umožňuje publikaci dokumentů na internetu. HTML značkám se též říká elementy.

Každá ze značek má nějaký význam. Například párová značka `<p>...</p>` označuje paragraf a přes `<img />` vkládáme do stránky obrázky. Mezi párové značky můžeme vkládat text nebo další HTML elementy.

## 3.4 Laravel

Laravel [1] je open-source PHP framework, který nabízí připravenou MVC architekturu a spoustu funkcionalit pro elegantní a jednoduchý vývoj webových aplikací.

### 3.4.1 MVC v Laravelu

MVC je architektura, která rozděluje aplikaci do 3 nezávislých komponent, které mezi sebou komunikují.

- **Model** reprezentuje informace, se kterými aplikace pracuje, 1 model = 1 sada souvisejících informací zapouzdřené v jednom objektu (modelem může být například hotel)
- **View** data z aplikace prezentuje v uživatelsky přívětivé formě, umožňuje uživateli interakci s nimi
- **Controller** reaguje na vstupy uživatele, zajišťuje změny v modelu a aktualizuje view

V Laravelu je MVC nastaveno stejně. Konkrétněji model je třída reprezentující jeden řádek tabulky, která je v něm definována přes proměnnou `$protected = 'table_name'`. Jsou v něm zároveň definované vztahy s dalšími modely. Model sice v Laravelu definuje jak vypadá řádek tabulky a dokonce i vlastnosti dané tabulky, ale nestará se o vytvoření takové tabulky. O to se starají migrace.

View jsou psána v *.blade* (*test.blade.php*) souborech. Zde můžeme psát HTML, PHP a využívat funkcí Laravelu. Ten zde definuje i speciální značky pro vkládání PHP kódu. `@php @endphp` místo `<?php ?>` značek a také kód, který provádí jednu operaci (například výpis proměnné) můžeme vložit mezi dvojité složené závorky `{{ kód }}` nebo při potřebě vygenerování HTML kódu `{!! kód !!}` . Poslední značky jsou obvzlášť nápomocné pro výpis proměnných do atributů HTML elementů.

```
<div data-name="{{ $userName }}">
  {!! $htmlCode !!}
</div>
```

Listing 3.1: Ukázka Laravel značek ve view

### 3.4.2 Databázové migrace

Migrace slouží ke kontrolovanému vývoji databáze aplikace. Každou jednotlivou migrací popisujeme jaké změny se mají v databázi provést (vytvoření tabulky nebo sloupce, úprava definice sloupce, apod.). V migraci také definujeme její zpětný chod , tzn. co se má stát, aby se databáze dostala do podoby před migrací.

```
    public function up() {
        Schema::create('users', function (Blueprint $table) {
```

```
            $table->bigIncrements('id'); // Primární klíč
            $table->string('username', 100); // 100 je velikost varchar v DB
            $table->unsignedBigInteger('user_role_id');
            $table->timestamp('created_at')
            ->default(DB::raw('CURRENT_TIMESTAMP'));

        // Cizí klíč
            $table->foreign('user_role_id')->references('id')->on('user_roles')
            ->onDelete('cascade');
        });
    }
```

Listing 3.2: Ukázka databázové migrace v Laravelu

## 3.5 CSS

CSS je jazyk, který popisuje, jak budou HTML elementy vypadat na stránce. Hlavním smyslem jazyka je oddělit vzhled dokumentu od jeho struktury a obsahu.

### 3.5.1 SCSS

SCSS je novější syntaxí CSS preprocesoru Sass, který dělá psaní stylů stránky mnohem elegantnější a přehlednější. Kód v SCSS se kompiluje do CSS kódu, který se pak může připnout ke stránce. Díky tomu také můžeme kód rozdělovat do více souborů a vytvářet stylové komponenty. Výsledný soubor pak bude ale stále jeden. Další výhodou je vnořování bloků elementů, díky tomu můžeme psát styly ještě přehledněji. V neposlední řádě SCSS umožňuje využívat proměnné, do kterých můžeme uložit jakékoliv hodnoty (pixely, barvy, atd.).

V následujících ukázkách můžeme vidět kódy v CSS a SCSS se stejným výstupem.

```
.my-class {
  display: block;
  width: 150px;
}

.my-class--smaller {
  width: 75px;
}
```

Listing 3.3: Ukázka kódu v CSS

```
$myClassWidth = 150px;

.my-class {
  display: block;
  width: $myClassWidth;

  &--smaller {
    width: $myClassWidth / 2;
  }
}
```

Listing 3.4: Ukázka kódu v SCSS

## 3.6 JavaScript

JS je multiplatformní skriptovací jazyk. Jedná se o jazyk, který se převážně používal pro vytvoření dynamických funkcionalit na webových stránkách. Ve většině případů se nespouští na straně serveru, ale na straně klienta.

V dnešní době zažívají JS a technologie na něm postavené velký rozmach. Už se nepoužívá pouze na straně klienta, ale existují technologie, díky kterých běží na straně serveru a tvoří tak back-end aplikací. Příkladem takovéto technologie je Node.js.

### 3.6.1 jQuery

jQuery je open-source javascriptová knihovna, která se zaměřuje na oddělení „chování“ stránky od HTML. Mimo to disponuje zjednodušenou syntaxí pro manipulaci s HTML a CSS. Pro získávání elementů z dokumentu využívá CSS selektorů.

V následujících ukázkách můžete vidět rozdíl syntaxe čistého JS a jQuery.

```
let element = document.getElementById('
    my-element');
element.style.color = 'white';
element.style.background-color = 'black;
```

Listing 3.5: Ukázka kódu v JS

```
$('#my-element').css({
   'color': 'white',
   'background-color': 'black'
});
```

Listing 3.6: Ukázka kódu v jQuery

## 3.7 Nginx

Nginx je open-source softwarový server určený pro hostování webových stránek a aplikací. Zaměřuje se na vysoký výkon a nízké nároky na paměť. Je jednou z možností webových serverů. Další je například Apache.

## 3.8 Docker

Docker je open-source projekt, který poskytuje rozhraní pro zapouzdření aplikací do tzv. kontejnerů nebo také obrazů. Jedná se o odlehčenou virtualizaci, kontejner obsahuje pouze požadované aplikace a pro ně potřebné soubory, neobsahuje operační systém, využívá ale Linuxového jádra systému, na kterém zrovna běží.

Pomáhá sjednodit vývojové prostředí pro každý projekt, tzn. zajistí, že každý vývojář stejného projektu bude mít stejné verze a nastavení prostředí (PHP, MySQL, Apache, atd.). Tím se elimi-

nuje složité instalování všech závislostí projektu a chybovost způsobená rozdílnými verzemi napříč platformami. Zároveň se tím zjednodušuje nahrávání projektů na vývojové a produkční prostředí.

Docker kontejnery můžeme uspořádat do sítí, ve kterých mezi sebou mohou komunikovat. Díky toho můžeme mít jeden kontejner pro aplikaci, další pro databázi, další pro server, atd.

## 3.9 Google Cloud Platform

Google Cloud Platform mimo jiné nabízí nástroje pro správu aplikací běžících na cloudu(v internetu). Zjednodušeně, Google nabízí „pronájem strojů“, na kterých se dají zprovoznit aplikace, které pak můžeme prezentovat světu. Přes GCP tyhle stroje a aplikace na nich spravujeme.

V případě našeho projektu využíváme GCP jako alternativu pro klasický hosting, jelikož díky tomu máme nad aplikací a jejím během větší kontrolu.

## 3.10 Git

Git je systém pro správu verzí projektů. Hlídá změny souborů a tyto změny pak lze posílat do vzdáleného repozitáře. K němu může mít přístup více uživatelů, ti si poslané změny mohou stáhnout k sobě lokálně a pokračovat tak ve vývoji projektu na stejném místě. Další výhodou je, že uchovává historii změn a může být využit pro nahrávání nových verzí projektů na vývojová prostředí, nemusí se tak dít ručně a díky tomu se minimalizuje chybovost.

## 3.11 Gitlab CI/CD

Gitlab je jedním z webových Git repozitářů. Mimo to ještě nabízí mnoho funkcionalit, pro efektivní práci s nimi a jejich využití pro další vylepšení vývojového procesu. Jedním z těchto rozšíření je Gitlab CI/CD.

Gitlab CI/CD je mimo jiné nástroj pro efektivní nahrávání změn v projektu na vývojová prostředí. Využívá virtuálních strojů, tzv. runner, na kterých dokáže spouštět uživatelem definované příkazy. Těmito příkazy uživatel může provést sestavení svojí aplikace a její následné nahrání na další vývojová prostředí.

# Kapitola 4

# Práce na projektu

V téhle sekci bude popsán projekt Spanamo a jednotlivé úkoly, které jsem na něm zpracovával.

| Úkol | Čas | |
|---|---|---|
| | v hodinách | ve dnech |
| Vývojový proces | 186 | 23 |
| Vyhledávání | 156 | 19 |
| Role a oprávnění | 118 | 15 |

Tabulka 4.1: Čas strávený na jednotlivých úkolech

## 4.1 Popis projektu

Jedná se o projekt, který se specializuje na zprostředkování lázeňských pobytů a procedur po celém světě. Skládá se ze 2 částí, které jsou obě postavené na frameworku Larave [1] a sdílejí stejnou databázi na MySQL.

**FRONT** je část určená zákazníkům. Na ní se prezentují všechny nabídky, ve kterých uživatel může vyhledávat a filtrovat. U konkrétních hotelů a pokojů pak vidí jejich fotogalerie, popisy vybavení, recenze a podmínky jejich rezervace. Při rezervaci si uživatel může k pobytu dokoupit i léčebné procedury. Jeho rezervace, hodnocení a oblíbené hotely si pak může prohlédnout ve svém profilu.

**SYSTEM** je administrativní část, ve které se spravují všechny nastavení projektu. Vytvářejí se zde lokality, hotely a jejich pokoje, včetně všech jejich informací a obrázků. Řeší se zde obsazenost a storno podmínky pokojů, dále je zde přehled objednávek, plateb a všech uživatelů. Uživatelé, kteří mohou do SYSTEMu vstupovat mají různé role a oprávnění, které jim umožňují nebo zabraňují vykonávat některé akce.

## 4.2 Zlepšení vývojového procesu

Při převzetí projektu dostalo Moravio pouze zdrojové kódy a vyexportovanou databázi. Bylo zapotřebí tedy projekt spustit lokálně a následně i na vývojovém prostředí (tzv. development prostředí nebo dev), ke kterému bude mít přístup i klient, aby mohl kontrolovat úpravy, zda odpovídají zadání, a plnit systém daty, které budou třeba při spuštění.

Při nastavení prvotního vývojového procesu byl projekt vložen do Git repozitáře na Gitlabu. Lokální prostředí se spoléhalo na PHP, MySQL a server spuštěných na stroji vývojáře. Na strojích s MacOS se nejčastěji používala aplikace MAMP, která všechny tyto technologie na počítači zprovozní (alternativa pro Windows je XAMPP).
Vývojové prostředí běželo na serveru vlastněným Moraviem, kde již všechny technologie potřebné pro běh projektu byly zprovozněné. Nahrávání změn bylo provedeno jednoduchým skriptem přes Gitlab CI/CD. Při poslání změny do repozitáře se přes tento skript stejná změna nahrála i na vývojovém prostředí.

### 4.2.1 Cíl úkolu

V průběhu vývoje jsme v týmu přišli na nevýhody takto řešeného procesu. Konkrétně ve chvíli, kdy se mělo implementovat fulltextové vyhledávání využívající vyhledávací modul Sphinx (tento úkol bude popsán později). Ten se totiž musel složitě zprovozňovat na stroji vývojáře, a s tím vyšly na povrch další problémy. Každý vývojář má o něco více nebo méně rozdílný stroj, na kterém pracuje, taktéž vývojové prostředí běží na úplně jiném stroji. Instalace modulu Sphinx se proto u každého prováděla trochu jinak a každý s tím měl jiné problémy. To nebylo ideální, protože to zbytečně zpomalovalo začlenění nových programátorů do projektu a celkově i jeho vývoj.

Cílem tohoto úkolu tedy bylo vymyslet a implementovat jednoduchý proces zprovoznění projektu na jakékoliv platformě. Moravio se zároveň začalo zbavovat svého development prostředí na cizím hostingu a přecházelo na řešení, které bude mít více pod kontrolou. Tím je Google Cloud [2]. Součástí úkolu se tedy stalo i zprovoznění projektu na Google Cloud a vytvoření efektivního nahrávání změn z Git repozitáře na něj.

### 4.2.2 Řešení

Jako ideální řešení pro sjednocení zprovoznění projektu napříč platformami a stroji byl vybrán **Docker** [3]. Všechny technologické závislosti, které projekt má, jsou v Docker obrazech, které si pak každý vývojář stáhne k sobě a pouze spustí. Tím se docílí toho, že každý bude mít stejné verze a nastavení všech závislostí a nebude muset složitě zprovozňovat každou z nich. Docker obrazy jde pak snadno stáhnout a spustit i na virtuálním stroji na Google Cloud a Gitlab CI/CD má nástroje pro práci s nimi.

#### 4.2.2.1 Vytvoření Docker obrazů projektu

Vytvoření Docker obrazu je proces, kdy se aplikace přesune do virtuálního stroje a zapouzdří do tzv. obrazu, který můžeme spouštět. Před započetím vytváření se musely obě části projektu připravit, jelikož každá měla být zapouzdřena zvlášť.
Jako první bylo třeba vytvořit soubor *.dockerignore*, ve kterém se specifikuje, jaké soubory a složky v obraze nechceme, většinou se jedná o konfigurační soubory a složky generované instalací interních závislostí projektu, apod. Další potřebnou úpravou je vytvoření konfiguračních souborů vázaných na prostředí (lokální, vývojové nebo produkční), které se pak samostatně do obrazu vloží. Pro Laravel je to například soubor *.env*, kde nalezneme základní nastavení aplikace, jako třeba její URL, přístupové údaje pro napojení na databázi, atd.

Po těchto přípravách může začít proces vytvoření obrazu. Ten se definuje v souboru ***Dockerfile*** [4], který jsme uložili do složky *docker* v kořenovém adresáři. V *Dockerfile* se píší příkazy, které se vykonávají ve virtuálním stroji. Jelikož Docker využívá Linux jádra systému, lze tedy využívat jakékoliv Linux příkazy pro práci uvnitř virtuálního stroje. Mimo ty, je možno ještě používat další speciální příkazy, které berou v potaz virtuální i fyzický stroj.
V následující sekci budou popsány krok po kroku nejdůležitější příkazy procesu. Jelikož je proces lehce rozdílný pro lokální prostředí a ostatní, bude popsán *Dockerfile* pro vývojové prostředí s informací, které příkazy se lokálně nevykonají.

##### 4.2.2.1.1 Důležité příkazy procesu

V této podsekci budou popsány nejdůležitější příkazy celého procesu.

- Vytváření začíná výběrem virtuálního stroje. Existují stroje, které již obsahují některé závislosti, které náš projekt potřebuje (například PHP). Výběr se provede následujícím příkazem, při jehož spuštění si systém stáhne z globálního Docker repozitáře vybraný obraz a všechny další příkazy spouští v něm.
  `$FROM php:7.2-fpm`

- Pokračuje se nakopírováním konfiguračních souborů určených pro cílové prostředí. `COPY` příkaz je speciálním příkazem, který dokáže kopírovat soubory z lokálního prostředí do virtuálního. Soubor *composer.json* a *.lock* obsahují seznam závislostí aplikace, které se mají nainstalovat, mimo jiné obsahují hlavně samotné soubory Laravel frameworku.
  Mimo konfigurační soubory Laravelu se zde mohou nakopírovat i konfigurační soubory PHP s upravenými hodnotami, například pro povolení objemnějších požadavků.
  V obou případech se neprovádí na lokálním prostředí.
  ```
  $COPY composer.lock composer.json /var/www/app/
  $COPY /docker/.env.devel /var/www/app/.env
  ```

```
$COPY /docker/.htaccess.public.devel /var/www/app/public/.htaccess
$COPY ./docker/app/remote/devel/local.ini $PHP_INI_DIR/conf.d/
```

- Dalším krokem je nastavení adresáře, ve kterém se budou ve virtuálním stroji spouštět další příkazy. Neprovádí se na lokálním prostředí.
  `$WORKDIR /var/www/app`

- Nyní je potřeba nainstalovat všechny systémové závislosti, které potřebuje naše aplikace, jedná se například o curl, který slouží pro stahování souborů přes počítačovou síť nebo konzolové rozhraní pro úpravu souborů Vim.
  `$RUN apt-get update && apt-get install -y curl zip vim`

- Jedním z posledních kroků je stáhnutí technologie pro správu závislostí Composer [5] a poté spustit samotnou instalaci závislostí. Neprovádí se na lokálním prostředí.
  ```
  $RUN curl -sS https://getcomposer.org/installer | php - -install-dir=/usr/local/bin
  -filename=composer
  $RUN composer install
  ```

- Poslední příkazy říkají na jakém portu ve virtuálním stroji aplikace poběží a poté se spustí PHP uvnitř obrazu.
  ```
  $EXPOSE 9000
  $CMD ["php-fpm"]
  ```

Při vytváření Docker obrazu aplikace pak mimo tento *Dockerfile* specifikujeme kořenový adresář aplikace a po úspěšném provedení všech příkazů máme k dispozici aplikaci, zapouzdřenou uvnitř obrazu, který můžeme libovolně spouštět a vypínat a poběží nezávisle na tom, zda je na stroji nainstalována správná verze PHP či nikoliv. Tento obraz je schopný manipulovat s dynamickými PHP soubory. Pro práci se statickými soubory, jako například obrázky, CSS a JS, je potřeba vytvořit další obraz této aplikace.

Podobným způsobem se vytvoří pro aplikaci obraz obsahující server, který bude zacházet se statickými soubory. Hlavním rozdílem oproti předešlému vytváření je výběr virtuálního stroje, kterým bude `ngninx:alpine` a místo instalování aplikačních závislostí přes Composer instalujeme balíčky, se kterými pracují SCSS a JS, přes NPM.

#### 4.2.2.2 Napojení obrazů na závislosti

Jakmile byly vytvořeny obrazy pro obě části projektu, musely být připojeny do sítě Docker obrazů. Síť se vytváří v konfiguračním souboru *docker-compose.yml* [6], který se píše v jazyce YAML.
První věc, která je zde definována je zdroj obrazů. Mohou být vytvořeny zadáním kontextu a *Dockerfile* nebo lze zadat URL na repozitář, kde jsou uloženy. Ukládání obrazu do repozitáře bude popsáno v sekci Gitlab CI/CD. Dále je zde popsáno, jak obrazy budou vypadat v rámci sítě a s jakými dalšími budou propojeny. Pokud tohle propojení není definováno, obrazy spolu navzájem nebudou moci komunikovat. V neposlední řadě se zde určuje, jaké složky na našem lokálním stroji se budou zrcadlit do jaké složky ve virtuálním prostředí, tím se zajistí perzistence složky. Nejčastěji se takto určují složky, kam se ukládají nahrávané soubory uživatelů aplikace. Pokud by tyto složky nebyly označeny jako perzistentní, při vypnutí a zapnutí sítě bychom přišli o všechny data, neboť se obrazy vždy spouští s čistým štítem.

```
app-front:
  image: registry.gitlab.com/moravio/thermaeurope/docker-laravel-phpfpm:latest
  container_name: app-front
  working_dir: /var/www/app
  links:
    - mysql
  volumes:
    - ../../sitemap-front.xml:/var/www/app/public/sitemap.xml:cached
    - ../../storage-front/:/var/www/app/storage/:cached
  networks:
    - thermaeurope-app-network
```

Listing 4.1: Ukázka definování obrazu v docker-compose.yml

V síti definujeme další obrazy.

- Nginx server, který poslouží jako směrovač dotazů mezi oběma částmi. V jeho konfiguraci je definováno, že bude poslouchat na portu 80, dále při zachycení domény spanamo.com bude dotaz přesměrován na FRONT a při zachycení sub-domény system.spanamo.com bude přesměrován na SYSTEM.
- Obraz se zprovozněnou MySQL databázi, na kterou bude napojen FRONT i SYSTEM.
- K tomu, abychom mohli do databáze přistupovat přes webové rozhraní je v síti i obraz obsahující adminer, přes který si můžeme databázi prohlížet.
- Sphinxsearch vyhledávací engine

Po nastavení sítě ji lze spustit konzolovým příkazem `$docker-compose up -d`, vlajka `-d` zde znamená, že chceme aby Docker nevyužíval konzoli, ve které se příkaz spouští, pro výpis běhu obrazů (z anglického detach neboli odpojit).

Síť lze vypnout příkazem `$docker-compose down`. Konfigurační soubor Docker sítě spolu s konfiguračními soubory všech obrazů sítě byl přidán do Git repozitáře.

#### 4.2.2.3 Zprovoznění projektu na Google Cloud

Před samotným zprovozněním si klient zařídil Google Cloud Platform [2] prostor a poskytl nám práva pro jeho správu, vše se řeší skrze Google účty. Prvním krokem bylo vybrání adekvátního stroje pro běh projektu, tedy definování kolik místa bude k dispozici pro instalaci všech systémových závislostí a zprovoznění projektu a jak silný stroj bude (kolik bude mít jader a výpočetní paměť). Jelikož se za každé místo, jádro a paměť platí, bylo třeba sestavit stroj tak, ať má akorát prostředků pro správný běh projektu.

Po nastavení stroje bylo na řadě zprovoznění projektu na tomto stroji, přes webovou konzoli. Prvně bylo třeba ale nainstalovat všechny systémové závislosti, které jsou potřebné pro běh projektu. Postup byl tedy podobný zprovoznění nového počítače, jelikož zde bylo zapotřebí nainstalovat všechny potřebné věci pro práci na něm. Pro nás to byl například Vim, zip, curl a hlavně Docker, docker-compose a Git.

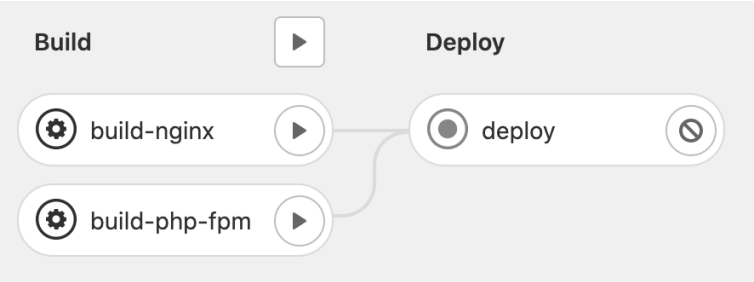
Jakmile byly nainstalovány všechny systémové závislosti, stáhl jsem přes Git konfigurační soubory pro Docker síť. Potom jsem se přesunul do složky s docker-compose.yml a spustil příkaz `$docker-compose pull`. Ten udělá to, že si stáhne všechny Docker obrazy, které konfigurační soubor obsahuje. Po tom, co dokončí stahování obrazů, je možno spustit síť. Po nasměrování domény na IP stroje na GCP aplikace běží a je přístupná z internetu.

#### 4.2.2.4 Gitlab CI/CD proces nahrávání

V tuhle chvíli jsem měl vytvořenou síť Docker obrazů a zprovozněné prostředí, na kterém bude aplikace dostupná veřejnosti. Posledním dílem skládačky bylo propojení lokálního vývojového prostředí s tím veřejným. Toho bylo docíleno přes Gitlab CI/CD [7].
Na Gitlabu byly samotné Git repozitáře obou částí projektu, bylo tedy příhodné využití nástrojů platformy pro nahrávání změn poslaných do těchto Git repozitářů na Google Cloud. Celý proces od nahrání úpravy až po její stáhnutí na Gogle Cloud se definuje v konfiguračním souboru *.gitlab-ci.yml*.

Proces je rozdělen do fází (neboli stages). Jednotlivé fáze mohou paralelně vykonávat více úkolů. Každý úkol vykonává skript ve virtuálním stroji, tzv. runner. Podobně jako u vytváření Docker obrazů si vybíráme virtuální stroj ve kterém se vytváření provádí, i zde si vybíráme stroj, ve kterém se provede námi napsaný skript. Dále se vybírá Git větev, ze které se nové změny budou brát a nahrávat. Proces se může spustit automaticky po nahrání změny dané Git větve nebo manuálně. Takto vypadá proces, kterého jsme chtěli dosáhnout.

V první fázi procesu se vytváří Docker obrazy dynamické i statické části aplikace. Po jejich vytvoření se nahrají do Docker registrů Gitlabu, odkud se mohou stáhnout na Google Cloud prostředí. Před započetím skriptu se virtuální stroj přihlásí do registrů platformy. Skript si poté stáhne aktu-




Obrázek 4.1: Proces nahrávání změn v Gitlab CI/CD

ální verzi obrazu, aby z ní mohl využít některé neměnné části a nemusel tak všechny kroky vytvoření provádět pokaždé znovu. Tyto neměnné části jsou uloženy v dočasné paměti (cache). Po úspěšném vytvoření obrazu jej stroj nahraje do registrů Gitlabu jako aktuální. Ve skriptu se využívají proměnné, které se definují v nastavení Gitlabu, ty se pak využívají voláním jejich jména, před které se vloží symbol $.

```
build-php-fpm:
  stage: build
  image: docker:19.03.13
  before_script:
    - echo -n $CI_JOB_TOKEN | docker login -u gitlab-ci-token --password-stdin
        $CI_REGISTRY
  script:
    - docker pull $CI_REGISTRY_IMAGE/docker-laravel-phpfpm:latest || true
    - docker build -f ./docker/app/remote/devel/Dockerfile --cache-from $CI_REGISTRY_IMAGE
        /docker-laravel-phpfpm:latest -t $CI_REGISTRY_IMAGE/docker-laravel-phpfpm .
    - docker push $CI_REGISTRY_IMAGE/docker-laravel-phpfpm
  only:
    - /^release.*$/
  allow_failure: false
  when: manual
```

Listing 4.2: Ukázka fáze nahrání obrazu v Gitlab CI/CD

Další fáze procesu se spustí po úspěšném provedení předchozí fáze. V ní se nahrané obrazy stáhnou z registrů do virtuálního stroje na Google Cloud. Aby tohle bylo možné, je třeba se s využitím GCP nástrojů připojit na onen stroj skrze SSH. K tomu jsem musel vygenerovat autorizační klíč v nastavení Google Cloud a uložit jej do proměnné `$GCLOUD_SERVICE_KEY`.
Jakmile se z virtuálního stroje na Gitlabu připojíme na Google cloud, spustil jsem tam skript, který se přesune do složky s *docker-compose.yml* a spustí příkaz `$docker-compose pull`, který stáhne aktuální obrazy v registrech Gitlabu do stroje. Každé takovéto stáhnutí obrazů zanechá po sobě staré verze obrazů. Posupem času se tyhle staré obrazy nahromadí a zaberou značnou část uložiště,

proto skript vždy před spuštěním stahování promaže nepoužívané obrazy. Tento skript v textové podobě je uložen v proměnné `$GCLOUD_PULL_SCRIPT`.
V následující ukázce lze vidět důležité změny v definici oproti fáze nahrání.

```
deploy:
  stage: deploy
  script:
    - echo $GCLOUD_SERVICE_KEY | base64 -d > ${HOME}/gcloud-service-key.json
    - gcloud auth activate-service-account --key-file ${HOME}/gcloud-service-key.json
    - gcloud config set project $GCLOUD_PROJECT_ID
    - gcloud compute ssh moravio_adm@instance-1 --zone=europe-west3-c --command="
        $GCLOUD_PULL_SCRIPT"
  when: on_success
```

Listing 4.3: Ukázka fáze stáhnutí obrazů v Gitlab CI/CD

### 4.2.3 Shrnutí procesu

Takto je proces zaveden téměř již od převzetí projektu a prokázal se jako velmi efektivní. Díky automatizaci nahrávání změn se programátoři nemusí zabývat složitým ručním nahráváním a mohou během něj pokračovat ve své práci.
Zprovoznění projektu v Docker síti umožnilo rychlé zapojení nových kolegů do projektu, neboť si nemusí složitě instalovat a nastavovat správné prostředí pro vývoj.
Proces by mohl být ještě dále optimalizován. Další vylepšení jsou zapotřebí hlavně z důvodů šetření financí, na Gitlabu se totiž platí za minuty, po které běží jednotlivé stroje. Jednou z možných optimalizací, které se v budoucnu pravděpodobně provedou, je vytváření pouze jednoho obrazu pro každou část projektu, nyní se vytváří dvě (PHP a Nginx).

Vytvoření tohoto procesu bylo náročnější než jsem původně odhadoval.

## 4.3 Nasazení fulltextového vyhledávání

Vyhledávání, tedy konkrétněji našeptávání ve vyhledávači lokalit a hotelů, je jedním z velmi důležitých prvků FRONT části projektu. Zákazníkovi, který zná svou vysněnou destinaci, dokáže tuto destinaci najít i přes pravděpodobné překlepy a neúplně zadaný hledaný výraz. Tomu, kdo neví jakou destinaci chce navštívit, nabídne relevantní výsledky i pro obecnější výrazy.

Při převzetí projektu bylo našeptávání již implementované, logika ale nebyla úplná a vyhledávání fungovalo přes dotazy na databázi s `LIKE %%` podmínky, tedy vybíráním těch řádků, jejichž sloupce obsahovaly hledaný výraz na jakémkoliv místě.
Takto nastavené vyhledávání bylo pomalé, jelikož dotazy do databáze trvaly poměrně dlouho a tím, že databáze byla už tak značně zatížená, další zátěž na ni při každém vyhledávání nebyla dobrým

řešením. Také psání logiky celého procesu bylo tím komplexnější.
Zároveň bylo třeba stávající logiku upravit a přidat do ní tzv. **alternativní slova**. Tyto slova, měla sloužit jako alternativy pro jiné slova (například překlepy) nebo pro konkrétní entity (hotely, lokality a země).

Nejefektivnějším nástrojem proti komplexitě je rozložení problému na více částí. Z toho důvodu byl jako alternativa pro původní hledání v databázi navrhnut fulltextový vyhledávací modul **Sphinx** [8]. Díky němu bylo možno oddělit procesy vyhledávání od databáze, tím zajistit rychlejší výsledky a snížit zatížení databáze.
Z důvodu nasazení nového způsobu našeptávání a potřeby nových funkcí, bylo mimo zprovoznění nového modulu mým úkolem také naprogramování celé logiky.

### 4.3.1 Sphinx

Sphinx je open-source vyhledávací knihovna, která dokáže efektivně hledat v textu. Funguje na principu tzv. indexů, to jsou struktury, které si Sphinx vygeneruje dle nastavených parametrů. Obsah indexů se určuje SQL dotazy na databázi. V indexech následně vyhledává zadané výrazy.
Podporuje tři z nejpoužívanějších databázových systémů, a to MariaDB, PostgreSQL a MySQL. V našem případě je napojen na naši MySQL databázi. Mimo to má připravené API pro populární jazyky jako PHP, Java nebo Python.

#### 4.3.1.1 Indexy

Jak již bylo zmíněno, indexy jsou tabulkám podobné struktury, vygenerované konkrétně `SELECT` dotazy na databázi. Sloupce popisují jaký typ záznamu do nich může být uložen a v pozdějším procesu můžeme každému sloupci dát jinou váhu pro výsledky. Každý řádek musí obsahovat unikátní ID a alespoň jeden záznam, ve kterém se může vyhledávat. Může také obsahovat jiné atributy, které pak mohou být použity pro dodatečné filtrování výsledků (například hodnocení hotelu).
Vyhledávání v indexech se provádí přes SphinxQL [9], což je SQL dialekt obohacený o práci s nimi.

```
source hotels {
   type      = mysql
   sql_host = mysql
   sql_user = root
  ...

   sql_query  = SELECT CRC32(CONCAT(h.slug, IFNULL(t.translation, ""))), \
                      h.slug, t.translation, h.hotel_star, h.id as id \
                    FROM hotels h \
                    LEFT JOIN translations t ON t.language_key = h.name_key \
                    GROUP BY id, t.translation
```

```
  sql_attr_uint = id
  sql_attr_uint = hotel_star
}

index h\right otels_index {
  source      = hotels
  min_infix_len    = 2
}
```

Listing 4.4: Ukázka definice Sphinx indexu

V projektu je definováno několik indexů. Hlavní obsahuje hotely se všemi potřebnými informacemi a překlady. Další obsahují země a lokality zemí. Poslední obsahuje alternativní slova.
Indexy jsou v projektu definovány standardně (viz. výpis 10.), ovšem mají jedno specifikum. Jak bylo již zmíněno, každý řádek musí mít unikátní ID, aby byl uložen do indexu. Při duplikovaných ID se uloží pouze první výsledek obsahující toto ID. To je značný problém pro indexy, které se skládají z dotazů využívající napojování tabulek, v takovém případě se stane, že téměř všechny záznamy pro jeden řádek zdrojové tabulky ignoruje i přes jejich celkovou unikátnost (rozdílné hodnoty v sloupcích).
Tohle se dá řešit hashováním všech sloupců řádku, výsledek hashování pak poslouží jako unikátní ID. Na jeho získání byla použita hashovací funkce `CRC32()` neboli 32-bitový Cyklický Redundantní Součet. V indexu je také definováno, kolik je třeba minimálně zadat písmen do hledaného výrazu, aby se spustilo vyhledávání (proměnná `min_prefix_len`).

#### 4.3.1.2 Napojení na projekt

Pro Sphinx existuje Docker obraz, ten byl připojen na síť obrazů projektu v souboru docker-compose.yml. Tímto způsobem mohl Sphinx komunikovat s databází a aplikace s ním. Aplikace s ním komunikuje skrze rozšíření do Laravelu využívající PHP SphinxAPI. Rozšíření se nainstaluje přes Composer [5] a poté se vytvoří konfigurační soubor, ve kterém se nadefinuje napojení na modul a jaké indexy jsou k dispozici. Indexy zde zároveň můžeme namapovat na tabulky v databázi (`'hotels_index' => ['table' => 'hotels', 'column' => 'id']`). Výsledky vyhledávání pak budou odpovídající řádky tabulky v Laravel struktuře Collection [10].

```
$sphinx = new SphinxSearch();
$results = $sphinx->search('italy hotel', 'hotels_index')
   ->limit(10)
   ->range('hotel_star', 3, 5)
   ->get();
```

Listing 4.5: Ukázka definice Sphinx indexu

### 4.3.2 Data pro vyhledávání

Před programováním logiky bylo třeba si určit jaké data budou pro vyhledávání potřeba. Nejvíce položek, ve kterých se vyhledává má index hotelů. Při vyhledávání hotelů totiž potřebujeme zároveň brát v potaz i jejich lokality a země. V něm se tedy mimo názvů hotelů uchovávají i názvy lokalit a států. Jelikož je web multijazyčný má jeden hotel v indexu více řádků, každý řádek obsahuje názvy v jednom jazyce. Posledním jeho sloupcem je adresa, která je taktéž v každém jazyce.

Ostatní indexy mají podobné hodnoty pro vyhledávání. Jeden je ale specifičtější, a to index alternativních slov. Alternativní slovo je zaprvé alternativní název pro entity, tedy hotely, lokality a země. Zadruhé může také zastupovat libovolný výraz, takové se nazývají globální alternativní slova. Jejich přiřazení může vypadat následovně. S těmito příklady se bude také pracovat dále.

- Hotelu „Hotel Italia“ přidáme alternativní slovo „htl italia“. Tohle slovo jej plně zastoupí ve vyhledávání. Tento příklad znázorňuje alternativní slovo entity.
- Itálii přidáme slovo entity „country“.
- Výrazu „hotel“ přidáme alternativu „qualche“ (což v italštině znamená cokoliv). Toto slovo bude taktéž ve vyhledávání plně zastupovat původní výraz. Těmto alternativám se říká globální.

Index obsahuje všechny tato slova.

Jelikož je vyhledávání možné i v Češtině a Ruštině, musely být do indexů přidány tabulky se speciálními znaky těchto jazyků.

### 4.3.3 Očekávané výsledky

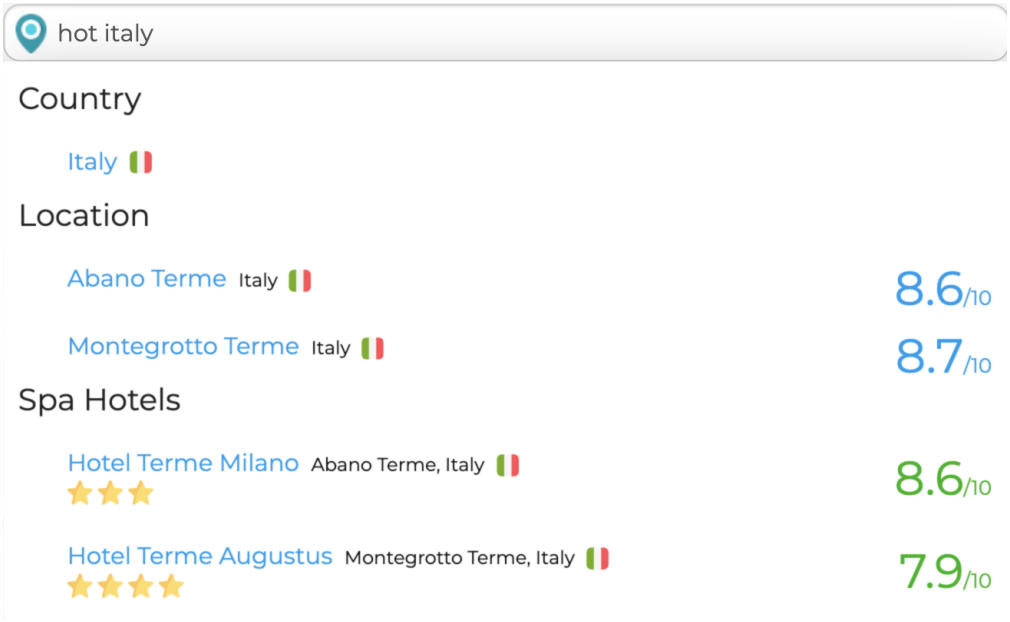
Pro lepší pochopení textu je zde přiložena ukázka vyhledávání (obr. 2.).

Při vyhledávání není třeba zadávat celé slova, ale stačí první dvě písmena. Ve výsledku jsou vždy obsaženy hotely, jejich lokality a země. Doplněné lokality a země nemusí být pouze výsledných hotelů, ale také mohou být samotným výsledkem vyhledávání. Mimo priority spojené s vyhledáváním ve Sphinx indexech se na výsledky aplikují i priority pro seřazení. Ty jsou zadávány v SYSTEMu u jednotlivých entit. Budou se podle nich seřazovat ale pouze doplněné (například státy hotelů a lokací), největší prioritu má vždy entita, která byla nalezena čistě zadaným výrazem.

Pokud bude nalezena entita z alternativního slova, použijeme dále ve vyhledávání i přeložený název této entity.

Při zadání výrazu „count htl“, by výsledek měl vypadat následovně.

- Bude nalezena Itálie přes alternativní slovo „country“. Dále ve vyhledávání se tedy použije název země v aktuálním jazyce aplikace.
- Bude nalezen hotel „Hotel Italia“ skrze alternativu „htl italia“. Bude to první výsledek u hotelů.




Obrázek 4.2: Ukázka výsledného vyhledávání

- Výsledné hotely budou doplněny těmi hotely, které budou mít nejlepší shodu s přeloženým jménem Itálie nebo „htl“. nejlepší shoda se urči počtem sloupců, ve kterých dané výrazy najde. Tyto hotel budou seřazeny dle zadaných priorit.
- Budou doplněny lokality všech hotelů ve výsledku a následně seřazeny dle zadaných priorit.

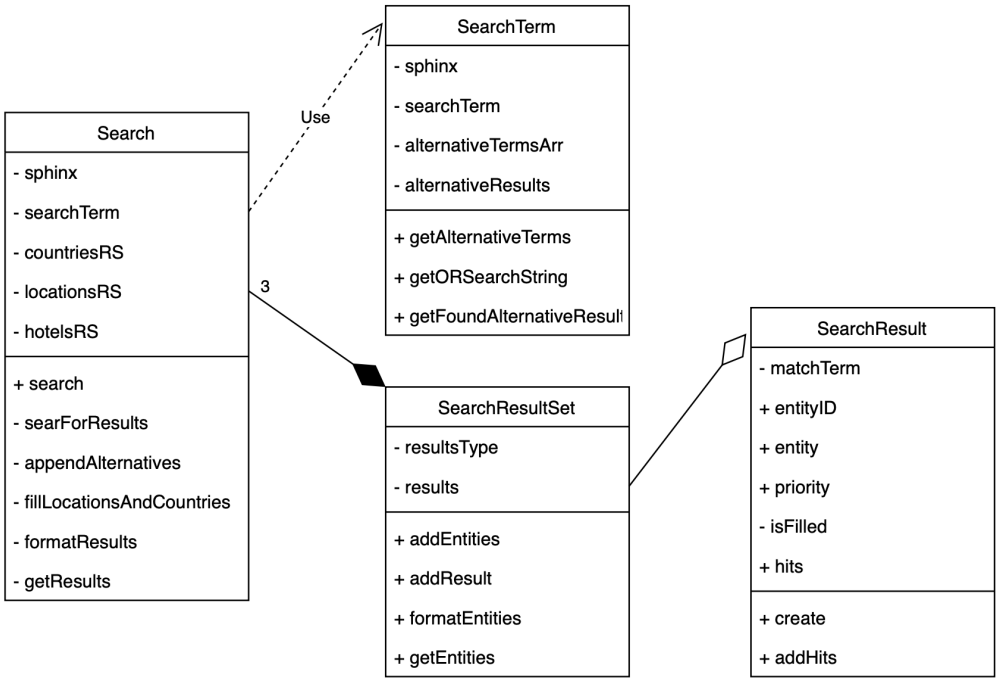
### 4.3.4 Implementace vyhledávání

V této podsekci bude popsáno řešení získání požadovaných výsledků. Vyhledávání se spustí po zadání výrazu do vyhledávacího pole. Přes JS se zavolá URL, na kterou se zašle hledaný výraz. Z té se zavolá Controller, který inicializuje třídu `Search` a spustí vyhledávání. Vyhledávání pak vrátí výsledky, které Controller vrátí v JSON formátu. Přes JS se pak výsledky vykreslí pod vyhledávacím polem.

#### 4.3.4.1 Třídy

Vyhledávací logika je zapouzdřena v několika třídách. Zde budou jednotlivé třídy popsány.

- `Search` je hlavní třída tohoto řešení.
- `SearchTerm` obstarává logiku kolem hledaného výrazu, formátuje jej a validuje, hledá jeho alternativy.




Obrázek 4.3: UML diagram tříd vyhledávání

- `SearchResult` reprezentuje výsledek vyhledávání. Obsahuje entitu a informace o tom, zda je doplněná a kolikrát byl přidávána do výsledků. Také obsahuje prioritu, nebo-li jak moc je významná (pokud byla nalezena originálním výrazem, má největší prioritu).

- `SearchResultSet` uchovává výsledky a ovládá jejich přidávání. Třída `Search` obsahuje tři tyhle třídy, každá pro jeden typ entity (hotel, lokalita, země).

#### 4.3.4.2 Logika

Vyhledávání se bude skládat z několika částí, každá z nich má pak několik kroků. V procesu se často využívá OR-LIKE notace pro vyhledávání v indexech. Ta funguje tak, že zadaný výraz se rozkouskuje pomocí mezer a jednotlivé slova se pak obalí v LIKE notaci `(*hot*)`, ta zajistí to, že nemusí být každé slovo zadáno celé. Takto obalená slova se pak sjednotí pomocí OR notace (použitím |), to zajistí, že nemusí být všechna slova přítomna u entity, stačí jen některá, zároveň ta entita, co bude mít nejvíce shod bude ohodnocena jako nejlepší výsledek. Výsledný OR-LIKE řetězec pak vypadá například takto: „`(*hot*)|(*it*)`“.

1. První částí je získání všech možných výrazů, se kterými se bude vyhledávat a následné jejich využití při vyhledávání. Každý z těchto výrazů bude mít prioritu, originální zadaný výraz má největší.

(a) Použitím třídy `SearchTerm` vyhledáme v indexu alternativních slov alternativní výrazy, se kterými pak budeme moct vyhledávat také. Pokud se při vyhledávání narazí na alternativní slova pro entity, tyto entity se uloží do třídy `SearchTerm` pro další použití.

(b) První z výrazů pro vyhledávání je samotný zadaný výraz v neupravené podobě a v OR-LIKE notaci.

(c) Další jsou nalezené alternativní výrazy v OR-LIKE notaci. A jako poslední jsou ty samé alternativní výrazy, akorát separované mezerami a dány do OR-LIKE podoby.

(d) Poté se vyhledá v indexech hotelů, lokalit a zemí. Všechny výsledky se přiřadí do příslušných `SearchResultSet`. Jako první budou entity, které byly nalezeny originálním výrazem.

2. Přidání dříve nalezených entit z alternativních výrazů a vyhledání v indexech pod jejich názvy.

   (a) Přidá tyto entity do příslušných `SearchResultSet`

   (b) Poté s jejich přeloženými názvy vyhledá znovu ve všech indexech a uloží entity.

   (c) Vyhledá pomocí alternativních výrazů v OR-LIKE notaci v indexu alternativních slov další entity, které pak přidá mezi výsledky.

3. V další části doplní lokality a země všech nalezených entit.

4. Všechny výsledné entity se zformátují, aby obsahovaly potřebné informace pro výpis (přeložený název, hodnocení a lokalitu).

5. Na závěr se výsledky seřadí podle počtu přidání do `SearchresultSet` a vrátí je.

#### 4.3.4.3 Přidávání entit do výsledků

Jednou z nejdůležitějších částí logiky je právě přidávání nalezených entit do výsledků v třídě `SearchResultSet`, musí se totiž mezi ně správně zařadit. Jelikož je možnost výsledek přidat před nebo za výsledky se stejnou prioritou, není možné entitu mezi ně jen přidat a pak kolekci seřadit podle priority, ale je třeba kolekci rozdělit na tři části. První část je s prioritou menší, druhá se stejnou a třetí s větší. To, jestli budeme entitu přidávat před nebo za ovlivní právě to, zda entitu vložíme před druhou část nebo za ní.
Pokud již entita ve výsledcích existuje, přidáme ji k zásahům jedničku.

```
public function addEntities(string $term, Collection $inputEntities, bool $fill, int
    $priority) {
  foreach($inputEntities as $entity)
    $this->addResult(new SearchResult($term, $fill, $entity, $priority));
}
```

```
public function addResult(SearchResult $newResult, $front = false) {
  if(!$this->results->contains("entityID", $newResult->entityID)) {
    $newCollection = $this->results->where('priority', '<', $newResult->priority)->all()
        ;
    if($front) $newCollection->push($newResult);
    $newCollection = $newCollection->merge($this->results->where('priority', '=',
        $newResult->priority)->all());
    if(!$front) $newCollection->push($newResult);
    $newCollection = $newCollection->merge($this->results->where('priority', '>',
        $newResult->priority)->all());

    if($newCollection != null) $this->results = $newCollection;
  } else {
    $this->results->where("entityID", $newResult->entityID)->first()->addHits(1);
  }
}
```

Listing 4.6: Zařazení entity do výsledků

### 4.3.5 Shrnutí úspěšnosti řešení

Přesunutím vyhledávání do modulu Sphinx se docílilo mnohem rychlejšího a relevantnějšího výsledku. Původní řešení bylo celé napsáno v jedné funkci Controlleru, rozdělením problematiky do více tříd se docílilo kódu, který je přehlednější a jednodušeji se upravuje. Řešení je možno ještě rozšířit o nastavování váhy jednotlivým sloupcům indexů pro přesnější výsledky.

Řešení jsem stihl provést ve stanoveném časovém odhadu i přes seznamování se s modulem Sphinx.

## 4.4 Uživatelské role a oprávnění

Do SYSTEMu bude mít přístup mnoho uživatelů, zaměstnanci klienta, tedy portálu a také zaměstnanci jednotlivých hotelů. Kvůli tomu, bylo třeba navrhnout modul pro správu jejich rolí a oprávnění v rámci SYSTEMu.

### 4.4.1 Cíl úkolu

V celém projektu existují čtyři uživatelské role, a to super administrátor, administrátor, hotel manažer a klient, nebo-li zákazník. Jedním z cílů tohoto úkolu bylo definování toho, jaké pravomoce bude mít každá z těch rolí v rámci SYSTEMu, tzn. jaké stránky budou moci navštěvovat a jak operace na nich budou moci provádět.

Dalším cílem bylo zanalyzování toho, jaké všechny operace může uživatel v aplikaci provádět. Na ty se pak musely vymyslet oprávnění, které by se mohly přidělovat rolím a uživatelům.

To vše bylo zároveň třeba naprogramovat a zprovoznit, konkrétně tedy logiku nastavování a kontrolování práv a rolí.

### 4.4.2 Návrh řešení

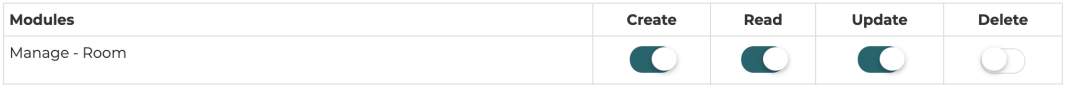
V této podsekci bude popsáno, jak byly navrženy jednotlivé části problému.

#### 4.4.2.1 Typy oprávnění

Analýzou bylo definováno několik druhů operací, pro které je zapotřebí vytvořit oprávnění. Konkrétně se jedná o:

- **„access“** se vystavují na konkrétní stránky v SYSTEMu a pokud jsou uživateli přiřazeny, je mu na ně povolen vstup. Obvykle se jedná o hlavní stránky modulů, které slouží jako rozcestníky pro další operace v rámci těch modulů.

- **CRUD** zahrnují hned čtyři operace, a to vytváření, čtení, úpravu a smazání. Tyhle oprávnění se napojují na modely Laravelu a říkají, zda uživatel může provádět dané operace nad jednotlivými modely. Každá z operací se povoluje samostatně, tzn. může se stát, že uživatel bude moct model číst, vytvářet a editovat, ale nebude jej moct mazat.




Obrázek 4.4: Ukázka nastavení CRUD oprávnění

- **„manage“** definují, nad jakými konkrétními entitami modelů mohou uživatelé provádět CRUD operace. Tyto oprávnění existují pouze pro modely hotelu, lokality a země. Pokud má uživatel přiřazený „manage“ pro hotel s ID 49, znamená to, že může nad tímto hotelem provádět všechny operace, které může provádět podle CRUD oprávnění. Zároveň ale také může díky těmto oprávněním provádět akce nad entitami hotelu s ID 49, tedy například spravovat pokoje hotelu, podle CRUD oprávnění, které má povolené u pokojů. Jiné hotely ale spravovat nemůže, i kdyby měl povoleny všechny oprávnění.

Oprávnění mohou být přiřazovány rolím i samotným uživatelů. Díky toho může mít například každý administrátor jiné oprávnění. Při vytváření uživatele, se mu nastaví oprávnění jeho role, potom mu ale mohou být nastaveny i ty které jeho role nemá.

#### 4.4.2.2 Uživatelské role

Role uživatelů jsou hierarchicky řazeny.

1. Super administrátor je nejvýše postavenou rolí. V reálném světe je jím klient samotný a jeho společníci. Tato role má neomezené práva, tzn. má přiřazené všechny oprávnění a může v SYSTEMu provádět všechny operace.

2. Administrátor je druhý v pořadí, jedná se většinou o zaměstnance portálu. Jeho oprávnění jsou nastavována. Může mít „manage“ oprávnění na hotely, lokality i země.

3. Hotel manažer je předposlední rolí, kterou budou mít přiřazeni zaměstnanci hotelů. Od administrátora se liší tím, že může mít „manage“ oprávnění pouze na hotely, nemůže tedy spravovat lokality a země.

4. Zákazník je nejnižší rolí v projektu. Do SYSTEMu nemá vůbec přístup. Spolu se super administrátorem jsou jedinými rolemi, které ale mohou přistupovat do uživatelského účtu ve FRONTu.

Hierarchické řazení pomáhá k řešení kolizí vzniklých CRUD oprávněními pro model uživatelů. Každému z uživatelů totiž může být přiřazeno oprávnění, které mu povoluje vytvářet, číst, editovat a mazat další uživatele. Bez jakéhokoliv ošetření by se mohlo stát, že si hotel manažer vytvoří účet super administrátora nebo, že administrátor smaže účet jinému administrátorovi, apod.
Proto se definovaly pravidla pro manipulaci s uživately napříč hierarchií.

1. Uživatel může vidět pouze uživatele na **stejné nebo nižší úrovn**i v hierarchii. Administrátor například ve výpise uživatelů nemůže vidět super administrátory.

2. Uživatel může vytvářet, editovat a mazat pouze ty uživatele, kteří jsou v hierarchii **postaveni níže**, tzn. administrátor nemůže tyto operace provádět nad jiným administrátorem, i když má pro to CRUD oprávnění povolené.

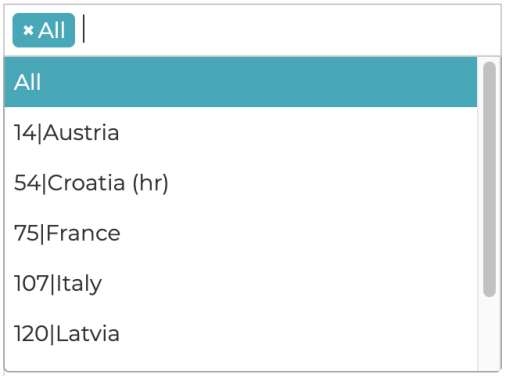
### 4.4.3 Řešení

Abychom nemuseli začínat řešení úplně od nuly, využili jsme Bouncer [11] rozšíření do Laravelu. Díky tomuto rozšíření jsme nemuseli řešit nízkoúrovňové záležitosti ukládání rolí a oprávnění a mohli jsme se zaměřit na naší logiku. Bouncer zároveň obsahuje mnoho pomocných funkcí pro kontrolu oprávnění, které má přímo napojené na ORM Laravelu. Mimo to už přichází s migracemi, které vytvoří tabulky uchovávající jednotlivé role a oprávnění a informace o tom, jaký uživatel má jaké pravomoce.

#### 4.4.3.1 Vytváření oprávnění

Každý typ oprávnění se vytváří svým unikátním způsobem.

- **„access“** vznikají v modulu menu, ve kterém se vytvářejí nové záložky do menu v SYSTEMu. Po vytvoření nové záložky se zároveň vygeneruje oprávnění, které pak může být přiřazeno uživatelům nebo rolím.
- **CRUD** se generují ručně. Bouncer [11], mimo již řečené věci, přichází i se „seederem“, to je soubor který po spuštění naplní databázi daty. Zde se definují například základní role a oprávnění, které by měl mít každý při spuštění projektu. My zde máme vytvořenou smyčku, která projde všechny modely vytvořené v SYSTEMu a pokud nejsou ignorovány, to určíme připravenou proměnnou, vygenerují se pro ně všechny čtyři CRUD oprávnění.
- **„manage“** se generují přímo v editaci uživatelů. Jsou zde pole s výběrem všech hotelů a v případě administrátora i lokalit a zemí. Pro každou vybranou entitu se po uložení vygeneruje „manage“ oprávnění, pokud již neexistuje. Zároveň se pak hned přiřadí danému uživateli.




Obrázek 4.5: Ukázka nastavení "manage"oprávnění

#### 4.4.3.2 Logika přiřazování oprávnění

Jako příklad použijeme editaci administrátora. Na stránce jeho editace se vypíší všechny možná oprávnění všech typů. CRUD a "access"jsou vyobrazeny jednoduchými přepínači, které reprezentují, zda má uživatel oprávnění přiřazené, či nikoliv (vzhled je k vidění na obr. 4.).

Oprávnění typu „manage“ jsou rozděleny do tří polí s nabídkou (jako obr. 5.). Jeden slouží pro výběr zemí, které bude moct spravovat, další pro lokality a poslední pro hotely. Pokud uživatel vybere u zemí například Itálii, ve výběru lokalit a hotelů se zobrazí pouze ty, které se nacházejí v Itálii. Obdobně je to s vybráním konkrétní lokality, po jejím vybrání se v nabídce hotelů zobrazí pouze ty, které se v ní nachází.
Získávání dat pro zobrazování validní nabídky probíhá skrze AJAX volání v JS. Skrze něj lze poslat

požadavek na libovolnou URL. Ta požadavek zpracuje a vrátí příslušné data, které se pak zobrazí pod zadávacím polem. V našem případě například u výběru hotelů AJAX zavolá routu, která zavolá funkci Controlleru a ta podle vybraných zemí a lokalit poslaných v požadavku vybere vhodné hotely, které vrátí ve formátu JSON.
Další specifikací tohoto typu je možnost vybrání „ALL“ oprávnění. To reprezentuje všechny entity, které jsou právě v nabídce. Jakmile si uživatel vybere jiné oprávnění, „ALL“ se zruší. Tak stejně, pokud si uživatel vybere „ALL“, vyruší se všechny ostatní konkrétní entity. Pokud je již vybrána nějaká země, tohle oprávnění v hotelech pak reprezentuje všechny hotely té země.

Samotný proces přiřazení oprávnění po uložení uživatele se skládá ze dvou částí. V datech, které putují do tohoto procesu se nachází všechny oprávnění, povolené i nepovolené mimo „manage“, ty se totiž vyobrazují dynamicky a nejsou tak přítomny ve formuláři, tedy mimo ty zvolené.

- V první části se „ALL“ převede na jednotlivé „manage“ oprávnění entit. Také se zde získají všechny dostupné „manage“ oprávnění v aplikaci, pro další použití.
- V druhé části se zvolené a povolené oprávnění uživateli přidělí a povolí, ty se zároveň vymažou ze všech dostupných oprávnění.
- Poté se projdou všechny zbylé oprávnění a uživateli se zakážou.

Celý proces probíhá se zapnutou dočasnou pamětí pro Bouncer, ten si tak ukládá výsledky dotazů na databázi, a tím se celkově zrychluje proces ukládání.

```
// Povolení oprávnění
Bouncer::allow($userOrRole)->to($operation, $model);
Bouncer::unforbid($userOrRole)->to($operation, $model);

// Zakázání oprávnění
Bouncer::disallow($userOrRole)->to($operation, $model);
Bouncer::forbid($userOrRole)->to($operation, $model);
```

Listing 4.7: Ukázka povolení a zakázání oprávnění

#### 4.4.3.3 Kontrola oprávnění

Každý z typu oprávnění se kontroluje jiným způsobem.

- **„access“** se kontroluje ještě před vstupem do funkce Controlleru. Každá URL menu se musí také definovat ručně v Laravelu jako routa. Routy mohou být seskupeny do skupin, kterým se pak mohou nastavit Middleware, což je třída s funkcí, která se spustí před tím, než se požadavek dostane do Controlleru. Všechny URL, které se mají kontrolovat pro tenhle typ oprávnění jsou seskupovány a samotná kontrola probíhá dynamicky právě v Middleware.

```
$page = DynamicMenu::where('link', implode('/', $request->segments()))->first();

if($page != null && !Auth::user()->can('access', $page))
   return $next($request);

return redirect('/unauthorized');
```

Listing 4.8: Kontrola "access"oprávnění

- **CRUD** oprávnění se kontrolují na konkrétních místech, kde chceme provést dané operace nebo chceme zobrazit element, který vede k jejich provedení.

```
@can('update', Media::class)
   <button class="btn" name="edit-btn">Editovat fotografii</button>
@encan
```

Listing 4.9: Kontrola CRUD oprávnění ve viewí

- **„manage“** se kontroluje převážně na začátku funkce v Controlleru, tedy ještě před tím, než se provedou nějaké akce nebo zobrazí data.

```
if(Auth::user()->cannot('manage', Hotel::find($request->hotel_id))
   return redirect('/unauthorized');
```

Listing 4.10: Kontrola "manage"oprávnění

Mimo jednotlivé typy oprávnění se také v každém požadavku kontroluje, zda uživatel, který chce vstoupit na danou stránku není klientem.

### 4.4.4 Hodnocení řešení

Implementované řešení převážně dynamickým způsobem pokrylo potřebu rozdělení uživatelů SYSTEMu a jejich pravomocí. Právě ona dynamičnost je největší výhodou, navedení nových oprávnění je velmi jednoduché, téměř automatické, a v případě „manage“ dokonce plně automatické. Kontrola oprávnění je jednoduchá a intuitivní.

Řešení má ale jednu nevýhodu, ukládání „manage“ oprávnění je pomalé, hlavně u administrátorů. Je to kvůli tomu, že uživateli se provede buďto operace povolení nebo zakázání oprávnění pro každou entitu a každá tato operace se skládá ze dvou databázových dotazů.
Řešením by mohlo být jiné zacházení s „ALL“, tedy že by se uživateli neukládali nutně všechny oprávnění, ale že by se mu například uložilo pouze „ALL“ hotelu a při kontrole by se v takovém případě kontrolovalo, zda daný hotel spadá do zemí a lokalit, které může uživatel spravovat.

V rámci řešení jsem musel provádět mnoho testování a byl kladen velký důraz na správnost řešení, kterého jsem nakonec dosáhl. Kvůli těmto důvodům se ale řešení protáhlo oproti plánovanému odhadu.

# Kapitola 5

# Závěr

## 5.1 Uplatněné znalosti a dovednosti nabyté při studiu

Díky praxe jsem si uvědomil, že jsem při studiu nabyl mnohých užitečných znalostí a dovedností. Nejvýznamnější bylo zažité přemýšlení nad problémem použitím tříd, které jsme se učili po celou dobu studia, především v předmětech Programování I a II, Programovací jazyky I a II, Skriptovací programovací jazyky a jejich aplikace a Tvorba aplikací pro mobilní zařízení II.
Díky předmětům Úvod do softwarového inženýrství a Vývoj informačních systémů jsem byl schopen navrhnout efektivní sestavení a význam tříd. Tyhle předměty mi ale zároveň pomohly rychle pochopit a správně použít architekturu Laravelu.
Na závěr bych chtěl vyzvednout i předměty Úvod do databázových systémů a Databázové a informační systémy, díky kterým jsem byl schopen pochopit komunikaci projektu s databází.

## 5.2 Nově nabyté znalosti a dovednosti praxí

Praxe všechny znalosti a dovednosti nabyté v době studia prohloubila jejich častým používáním a vystavením reálným problémům. Ukázala mi, že dobrého programátora nedělá pouze jím správně napsaný kód, ale také náležitě okomentovaný, zdokumentovaný a otestovaný.
Tam, kde mé znalosti nestačily, byli vždy ke konzultaci problémů i návrhu řešení k dispozici zkušenější seniorní kolegové, kteří mi ochotně pomohli. Tím jsme pokaždé zajistili požadovanou kvalitu pro klienta a zároveň se mi tím dostalo pomoci v rozvoji mých praktických zkušeností.
Mimo to mi dala praxe obrovský vhled do toho, jak funguje vývojářská firma a díky pravidelným firemním statusům i vhled do celého IT trhu. V neposlední řadě jsem se také naučil lépe komunikovat v týmu a také s klientem.

# Literatura


1. *Documentation for the Laravel* [online] [cit. 2021-03-16]. Dostupné z: `https://laravel.com/docs/5.8`.
2. *Documentation for the Google Cloud Platform* [online] [cit. 2021-03-16]. Dostupné z: `https://cloud.google.com/docs`.
3. *Documentation for the Docker* [online] [cit. 2021-03-16]. Dostupné z: `https://docs.docker.com/`.
4. *Documentation for the Dockerfile* [online] [cit. 2021-03-16]. Dostupné z: `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`.
5. *Compose informationsr* [online] [cit. 2021-03-16]. Dostupné z: `https://getcomposer.org/`.
6. *Documentation for the docker-compose* [online] [cit. 2021-03-16]. Dostupné z: `https://docs.docker.com/engine/reference/commandline/compose/`.
7. *Documentation for the Gitlab CI/CD* [online] [cit. 2021-03-16]. Dostupné z: `https://docs.gitlab.com/ee/ci/`.
8. *Documentation for the Sphinx search engine* [online] [cit. 2021-03-16]. Dostupné z: `http://sphinxsearch.com/docs/sphinx3.html`.
9. *Documentation for the SphinxQL syntax* [online] [cit. 2021-03-16]. Dostupné z: `http://sphinxsearch.com/docs/sphinx3.html#searching-query-syntax`.
10. *Documentation for the Laravel Collection* [online] [cit. 2021-03-16]. Dostupné z: `https://laravel.com/docs/5.8/eloquent-collections`.
11. *Documentation for the Laravel package Bouncer* [online] [cit. 2021-03-16]. Dostupné z: `https://github.com/JosephSilber/bouncer/tree/v1.0.0-rc.6`.